Chapter 7 Fundamentals of Input and Output

7-1. Displaying Information On-Screen

You will want most of your programs to display information on-screen. The two most frequently used ways to do this are with C's library functions printf() and puts().

7-1-1. The printf() Function

The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on-screen. You've already seen printf() used in many of the examples in this book. Now you will to see how printf() works.

Printing a text message on-screen is simple. Call the printf() function, passing the desired message enclosed in double quotation marks. For example, to display An error has occurred! on-screen, you write

```
printf("An error has occurred!");
```

In addition to text messages, however, you frequently need to display the value of program variables. This is a little more complicated than displaying only a message. For example, suppose you want to display the value of the numeric variable x on-screen, along with some identifying text. Furthermore, you want the information to start at the beginning of a new line. You could use the printf() function as follows:

```
printf("\nThe value of x is %d", x);
```

The resulting screen display, assuming that the value of x is 12, would be

```
The value of x is 12
```

In this example, two arguments are passed to printf(). The first argument is enclosed in double quotation marks and is called the *format string*. The second argument is the name of the variable (x) containing the value to be printed.

7-1-2. The printf() Format Strings

A printf() format string specifies how the output is formatted. Here are the three possible components of a format string:

- *Literal text* is displayed exactly as entered in the format string. In the preceding example, the characters starting with the T (in The) and up to, but not including, the % comprise a literal string.
- An *escape sequence* provides special formatting control. An escape sequence consists of a backslash (\) followed by a single character. In the preceding example, \n is an escape sequence. It is called the *newline character,* and it means "move to the start of the next line." Escape sequences are also used to print certain characters. Escape sequences are listed in Table 7.1.
- A *conversion specifier* consists of the percent sign (%) followed by a single character. In the example, the conversion specifier is %d. A conversion specifier tells printf() how to interpret the variable(s) being printed. The %d tells printf() to interpret the variable x as a signed decimal integer.

Table 7.1. The most frequently used escape sequences.

Sequence	Meaning
\a	Bell (alert)
\b	Backspace

\n	Newline
\t	Horizontal tab
\\	Backslash
\?	Question mark
\'	Single quotation

The printf() Escape Sequences

Now let's look at the format string components in more detail. Escape sequences are used to control the location of output by moving the screen cursor. They are also used to print characters that would otherwise have a special meaning to printf(). For example, to print a single backslash character, include a double backslash (\\) in the format string. The first backslash tells printf() that the second backslash is to be interpreted as a literal character, not as the start of an escape sequence. In general, the backslash tells printf() to interpret the next character in a special manner. Here are some examples:

Sequence	*Meaning*
N	The character n
\n	Newline
\"	The double quotation character
"	The start or end of a string

Table 7.1 lists C's most commonly used escape sequences. A full list can be found on Day 15, "Pointers: Beyond the Basics."

Listing 7.1 demonstrates some of the frequently used escape sequences.

Listing 7.1. Using printf() escape sequences.

```
1:   /* Demonstration of frequently used escape sequences */
2:
3:   #include <stdio.h>
4:
5:   #define QUIT  3
6:
7:   int  get_menu_choice( void );
8:   void print_report( void );
9:
10:  main()
11:  {
12:      int choice = 0;
13:
14:      while (choice != QUIT)
15:      {
16:         choice = get_menu_choice();
17:
18:         if (choice == 1)
19:             printf("\nBeeping the computer\a\a\a" );
20:         else
21:         {
```

```
22:             if (choice == 2)
23:                 print_report();
24:         }
25:     }
26:     printf("You chose to quit!\n");
27:
28:     return 0;
29: }
30:
31: int get_menu_choice( void )
32: {
33:     int selection = 0;
34:
35:     do
36:     {
37:         printf( "\n" );
38:         printf( "\n1 - Beep Computer" );
39:         printf( "\n2 - Display Report");
40:         printf( "\n3 - Quit");
41:         printf( "\n" );
42:         printf( "\nEnter a selection:" );
43:
44:         scanf( "%d", &selection );
45:
46:      }while ( selection < 1 || selection > 3 );
47:
48:      return selection;
49: }
50:
51: void print_report( void )
52: {
53:      printf( "\nSAMPLE REPORT" );
54:      printf( "\n\nSequence\tMeaning" );
55:      printf( "\n=========\t=======" );
56:      printf( "\n\\a\t\tbell (alert)" );
57:      printf( "\n\\b\t\tbackspace" );
58:      printf( "\n...\t\t...");
59: }
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
=========       =======
\a              bell (alert)
```

```
\b              backspace
...             ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

ANALYSIS: Listing 7.1 seems long compared with previous examples, but it offers some additions that are worth noting. The STDIO.H header was included in line 3 because printf() is used in this listing. In line 5, a constant named QUIT is defined. From Day 3, "Storing Data: Variables and Constants," you know that #define makes using the constant QUIT equivalent to using the value 3. Lines 7 and 8 are function prototypes. This program has two functions: get_menu_choice() and print_report(). get_menu_choice() is defined in lines 31 through 49. This is similar to the menu function in Listing 6.5. Lines 37 and 41 contain calls to printf() that print the newline escape sequence. Lines 38, 39, 40, and 42 also use the newline escape character, and they print text. Line 37 could have been eliminated by changing line 38 to the following:

```
printf( "\n\n1 - Beep Computer" );
```

However, leaving line 37 makes the program easier to read.

Looking at the main() function, you see the start of a while loop on line 14. The while loop's statements will keep looping as long as choice is not equal to QUIT. Because QUIT is a constant, you could have replaced it with 3; however, the program wouldn't be as clear. Line 16 gets the variable choice, which is then analyzed in lines 18 through 25 in an if statement. If the user chooses 1, line 19 prints the newline character, a message, and then three beeps. If the user selects 2, line 23 calls the function print_report().

print_report() is defined on lines 51 through 59. This simple function shows the ease of using printf() and the escape sequences to print formatted information to the screen. You've already seen the newline character. Lines 54 through 58 also use the tab escape character, \t. It aligns the columns of the report vertically. Lines 56 and 57 might seem confusing at first, but if you start at the left and work to the right, they make sense. Line 56 prints a newline (\n), then a backslash (\), then the letter a, and then two tabs (\t\t). The line ends with some descriptive text, (bell (alert)). Line 57 follows the same format.

This program prints the first two lines of Table 7.1, along with a report title and column headings. In exercise 9 at the end of this chapter, you will complete this program by making it print the rest of the table.

The printf() Conversion Specifiers

The format string must contain one conversion specifier for each printed variable. printf() then displays each variable as directed by its corresponding conversion specifier. You'll learn more about this process on Day 15. For now, be sure to use the conversion specifier that corresponds to the type of variable being printed.

Exactly what does this mean? If you're printing a variable that is a signed decimal integer (types int and long), use the %d conversion specifier. For an unsigned decimal integer (types unsigned int and unsigned long), use %u. For a floating-point variable (types float and double), use the %f specifier. The conversion specifiers you need most often are listed in Table 7.2.

Table 7.2. The most commonly needed conversion specifiers.

Specifier	Meaning	Types Converted
%c	Single character	char
%d	Signed decimal integer	int, short
%ld	Signed long decimal integer	long

%f	Decimal floating-point number	float, double
%s	Character string	char arrays
%u	Unsigned decimal integer	unsigned int, unsigned short
%lu	Unsigned long decimal integer	unsigned long

The literal text of a format specifier is anything that doesn't qualify as either an escape sequence or a conversion specifier. Literal text is simply printed as is, including all spaces.

What about printing the values of more than one variable? A single printf() statement can print an unlimited number of variables, but the format string must contain one conversion specifier for each variable. The conversion specifiers are paired with variables in left-to-right order. If you write

```
printf("Rate = %f, amount = %d", rate, amount);
```

the variable rate is paired with the %f specifier, and the variable amount is paired with the %d specifier. The positions of the conversion specifiers in the format string determine the position of the output. If there are more variables passed to printf() than there are conversion specifiers, the unmatched variables aren't printed. If there are more specifiers than variables, the unmatched specifiers print "garbage."

You aren't limited to printing the value of variables with printf(). The arguments can be any valid C expression. For example, to print the sum of x and y, you could write

```
z = x + y;
printf("%d", z);
```

You also could write

```
printf("%d", x + y);
```

Any program that uses printf() should include the header file STDIO.H. Listing 7.2 demonstrates the use of printf(). Day 15 gives more details on printf().

Listing 7.2. Using printf() to display numerical values.

```
1:   /* Demonstration using printf() to display numerical values. */
2:
3:   #include <stdio.h>
4:
5:   int a = 2, b = 10, c = 50;
6:   float f = 1.05, g = 25.5, h = -0.1;
7:
8:   main()
9:   {
10:      printf("\nDecimal values without tabs: %d %d %d", a, b, c);
11:      printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b,
c);
12:
13:      printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);
14:      printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f, g,
h);
15:
16:      printf("\nThe rate is %f%%", f);
```

```
17:      printf("\nThe result of %f/%f = %f\n", g, f, g / f);
18:
19:      return 0;
20:  }
Decimal values without tabs: 2 10 50
Decimal values with tabs:       2       10      50
Three floats on 1 line:         1.050000        25.500000       -
0.100000
Three floats on 3 lines:
        1.050000
        25.500000
        -0.100000
The rate is 1.050000%
The result of 25.500000/1.050000 = 24.285715
```

ANALYSIS: Listing 7.2 prints six lines of information. Lines 10 and 11 each print three decimals: a, b, and c. Line 10 prints them without tabs, and line 11 prints them with tabs. Lines 13 and 14 each print three float variables: f, g, and h. Line 13 prints them on one line, and line 14 prints them on three lines. Line 16 prints a float variable, f, followed by a percent sign. Because a percent sign is normally a message to print a variable, you must place two in a row to print a single percent sign. This is exactly like the backslash escape character. Line 17 shows one final concept. When printing values in conversion specifiers, you don't have to use variables. You can also use expressions such as g / f, or even constants.

DON'T try to put multiple lines of text into one printf() statement. In most instances, it's clearer to print multiple lines with multiple print statements than to use just one with several newline (\n) escape characters.

DON'T forget to use the newline escape character when printing multiple lines of information in separate printf() statements.

DON'T misspell stdio.h. Many C programmers accidentally type studio.h; however, there is no u.

The printf() Function

```
#include <stdio.h>
printf( format-string[,arguments,...]);
```

printf() is a function that accepts a series of *arguments*, each applying to a conversion specifier in the given format string. printf() prints the formatted information to the standard output device, usually the display screen. When using printf(), you need to include the standard input/output header file, STDIO.H.

The *format-string* is required; however, arguments are optional. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly used conversion specifiers.

The *format-string* can also contain escape sequences. Table 7.1 lists the most frequently used escape sequences.

The following are examples of calls to printf() and their output:

Example 1 Input

```
#include <stdio.h>
```

```
main()
{
    printf("This is an example of something printed!");
    return 0;
}
```

Example 1 Output

```
This is an example of something printed!
```

Example 2 Input

```
printf("This prints a character, %c\na number, %d\na floating point,
%f", `z', 123, 456.789 );
```

Example 2 Output

```
This prints a character, z
a number, 123
a floating point, 456.789
```

7-1-3. Displaying Messages with puts()

The puts() function can also be used to display text messages on-screen, but it can't display numeric variables. puts() takes a single string as its argument and displays it, automatically adding a newline at the end. For example, the statement

```
puts("Hello, world.");
```

performs the same action as

```
printf("Hello, world.\n");
```

You can include escape sequences (including \n) in a string passed to puts(). They have the same effect as when they are used with printf() (see Table 7.1).

Any program that uses puts() should include the header file STDIO.H. Note that STDIO.H should be included only once in a program.

DO use the puts() function instead of the printf() function whenever you want to print text but don't need to print any variables.

DON'T try to use conversion specifiers with the puts() statement.

The puts() Function

```
#include <stdio.h>
```

```
puts( string );
```

puts() is a function that copies a string to the standard output device, usually the display screen. When you use puts(), include the standard input/output header file (STDIO.H). puts() also appends a newline character to the end of the string that is printed. The for- mat string can contain escape sequences. Table 7.1 lists the most frequently used escape sequences.

The following are examples of calls to puts() and their output:

Example 1 Input

```
puts("This is printed with the puts() function!");
```

Example 1 Output

```
This is printed with the puts() function!
```

Example 2 Input

```
puts("This prints on the first line. \nThis prints on the second
line.");
puts("This prints on the third line.");
puts("If these were printf()s, all four lines would be on two
lines!");
```

Example 2 Output

```
This prints on the first line.
This prints on the second line.
This prints on the third line.
If these were printf()s, all four lines would be on two lines!
```

7-2. Inputting Numeric Data with scanf()

Just as most programs need to output data to the screen, they also need to input data from the keyboard. The most flexible way your program can read numeric data from the keyboard is by using the scanf() library function.

The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() uses a format string to describe the format of the input. The format string utilizes the same conversion specifiers as the printf() function. For example, the statement

```
scanf("%d", &x);
```

reads a decimal integer from the keyboard and assigns it to the integer variable x. Likewise, the following statement reads a floating-point value from the keyboard and assigns it to the variable rate:

```
scanf("%f", &rate);
```

What is that ampersand (&) before the variable's name? The & symbol is C's *address-of* operator, which is fully explained on Day 9, "Understanding Pointers." For now, all you need to remember is that scanf() requires the & symbol before each numeric variable name in its argument list (unless the variable is a pointer, which is also explained on Day 9).

A single scanf() can input more than one value if you include multiple conversion specifiers in the format string and variable names (again, each preceded by & in the argument list). The following statement inputs an integer value and a floating-point value and assigns them to the variables x and rate, respectively:

```
scanf("%d %f", &x, &rate);
```

When multiple variables are entered, scanf() uses white space to separate input into fields. White space can be spaces, tabs, or new lines. Each conversion specifier in the scanf() format string is matched with an input field; the end of each input field is identified by white space.

This gives you considerable flexibility. In response to the preceding scanf(), you could enter

```
10 12.45
```

You also could enter this:

```
10                  12.45
```

or this:

```
10
12.45
```

As long as there's some white space between values, scanf() can assign each value to its variable.

As with the other functions discussed in this chapter, programs that use scanf() must include the STDIO.H header file. Although Listing 7.3 gives an example of using scanf(), a more complete description is presented on Day 15.

Listing 7.3. Using scanf() to obtain numerical values.

```
1:   /* Demonstration of using scanf() */
2:
3:   #include <stdio.h>
4:
5:   #define QUIT 4
6:
7:   int get_menu_choice( void );
8:
9:   main()
10:  {
11:      int   choice    = 0;
12:      int   int_var   = 0;
13:      float float_var = 0.0;
14:      unsigned unsigned_var = 0;
15:
16:      while (choice != QUIT)
17:      {
18:          choice = get_menu_choice();
19:
20:          if (choice == 1)
21:          {
22:              puts("\nEnter a signed decimal integer (i.e. -123)");
23:              scanf("%d", &int_var);
```

```
24:          }
25:          if (choice == 2)
26:          {
27:              puts("\nEnter a decimal floating-point number\
28                   (i.e. 1.23)");
29:              scanf("%f", &float_var);
30:          }
31:          if (choice == 3)
32:          {
33:               puts("\nEnter an unsigned decimal integer \
34                   (i.e. 123)" );
35:               scanf( "%u", &unsigned_var );
36:          }
37:      }
38:      printf("\nYour values are: int: %d  float: %f  unsigned: %u
\n",
39:                                int_var, float_var, unsigned_var
);
40:
41:     return 0;
42:  }
43:
44:  int get_menu_choice( void )
45:  {
46:      int selection = 0;
47:
48:      do
49:      {
50:          puts( "\n1 - Get a signed decimal integer" );
51:          puts( "2 - Get a decimal floating-point number" );
52:          puts( "3 - Get an unsigned decimal integer" );
53:          puts( "4 - Quit" );
54:          puts( "\nEnter a selection:" );
55:
56:          scanf( "%d", &selection );
57:
58:       }while ( selection < 1 || selection > 4 );
59:
60:       return selection;
61:  }
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
```

```
4 - Quit
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned: 321
```

ANALYSIS: Listing 7.3 uses the same menu concepts that were used in Listing 7.1. The differences in get_menu_choice() (lines 44 through 61) are minor but should be noted. First, puts() is used instead of printf(). Because no variables are printed, there is no need to use printf(). Because puts() is being used, the newline escape characters have been removed from lines 51 through 53. Line 58 was also changed to allow values from 1 to 4 because there are now four menu options. Notice that line 56 has not changed; however, now it should make a little more sense. scanf() gets a decimal value and places it in the variable selection. The function returns selection to the calling program in line 60.

Listings 7.1 and 7.3 use the same main() structure. An if statement evaluates choice, the return value of get_menu_choice(). Based on choice's value, the program prints a message, asks for a number to be entered, and reads the value using scanf(). Notice the difference between lines 23, 29, and 35. Each is set up to get a different type of variable. Lines 12 through 14 declare variables of the appropriate types.

When the user selects Quit, the program prints the last-entered number for all three types. If the user didn't enter a value, 0 is printed, because lines 12, 13, and 14 initialized all three types. One final note on lines 20 through 36: The if statements used here are not structured well. If you're thinking that an if...else structure would have been better, you're correct. Day 14, "Working with the Screen, Printer, and Keyboard," introduces a new control statement, switch. This statement offers an even better option.

DON'T forget to include the address-of operator (&) when using scanf() variables.

DO use printf() or puts() in conjunction with scanf(). Use the printing functions to display a prompting message for the data you want scanf() to get.

The scanf() Function

```
#include <stdio.h>
scanf( format-string[,arguments,...]);
```

scanf() is a function that uses a conversion specifier in a given format-string to place values into variable arguments. The arguments should be the addresses of the variables rather than the actual variables themselves. For numeric variables, you can pass the address by putting the address-of operator (&) at the beginning of the variable name. When using scanf(), you should include the STDIO.H header file.

scanf() reads input fields from the standard input stream, usually the keyboard. It places each of these read fields into an argument. When it places the information, it converts it to the format of the corresponding specifier in the format string. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly needed conversion specifiers.

Example 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

Example 2

```
#include <stdio.h>
main()
{
    float y;
    int x;
    puts( "Enter a float, then an int" );
    scanf( "%f %d", &y, &x);
    printf( "\nYou entered %f and %d ", y, x );
    return 0;
}
```

7-3. Summary

With the completion of this chapter, you are ready to write your own C programs. By combining the printf(), puts(), and scanf() functions and the programming control statements you learned about in earlier chapters, you have the tools needed to write simple programs.

Screen display is performed with the printf() and puts() functions. The puts() function can display text messages only, whereas printf() can display text messages and variables. Both functions use escape sequences for special characters and printing controls.

The scanf() function reads one or more numeric values from the keyboard and interprets each one according to a conversion specifier. Each value is assigned to a program variable.

Q&A

Q Why should I use puts() if printf() does everything puts() does and more?
A Because printf() does more, it has additional overhead. When you're trying to write a small, efficient program, or when your programs get big and resources are valuable, you will want to take advantage of the smaller overhead of puts(). In general, you should use the simplest available resource.
Q Why do I need to include STDIO.H when I use printf(), puts(), or scanf()?
A STDIO.H contains the prototypes for the standard input/output functions. printf(), puts(), and scanf() are three of these standard functions. Try running a program without the STDIO.H header and see the errors and warnings you get.
Q What happens if I leave the address-of operator (&) off a scanf() variable?
A This is an easy mistake to make. Unpredictable results can occur if you forget the address-of operator. When you read about pointers on Days 9 and 13, you will understand this better. For now, know that if you omit the address-of operator, scanf() doesn't place the entered information in your variable, but in some

other place in memory. This could do anything from apparently having no effect to locking up your computer so that you must reboot.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the difference between puts() and printf()?
2. What header file should you include when you use printf()?
3. What do the following escape sequences do?
a. \\
b. \b
c. \n
d. \t
e. \a
4. What conversion specifiers should be used to print the following?
a. A character string
b. A signed decimal integer
c. A decimal floating-point number
5. What is the difference between using each of the following in the literal text of puts()?
a. b
b. \b
c. \
d. \\

Exercises

NOTE: Starting with this chapter, some of the exercises ask you to write complete programs that perform a particular task. Because there is always more than one way to do things in C, the answers provided at the back of the book shouldn't be interpreted as the only correct ones. If you can write your own code that performs what's required, great! If you have trouble, refer to the answer for help. The answers are presented with minimal comments because it's good practice for you to figure out how they operate.

---

1. Write both a printf() and a puts() statement to start a new line.
2. Write a scanf() statement that could be used to get a character, an unsigned decimal integer, and another single character.
3. Write the statements to get an integer value and print it.
4. Modify exercise 3 so that it accepts only even values (2, 4, 6, and so on).
5. Modify exercise 4 so that it returns values until the number 99 is entered, or until six even values have been entered. Store the numbers in an array. (Hint: You need a loop.)
6. Turn exercise 5 into an executable program. Add a function that prints the values, separated by tabs, in the array on a single line. (Print only the values that were entered into the array.)
7. BUG BUSTER: Find the error(s) in the following code fragment:

```
printf( "Jack said, "Peter Piper picked a peck of pickled
peppers."");
```

8. BUG BUSTER: Find the error(s) in the following program:

```
int get_1_or_2( void )
{
    int answer = 0;
```

```
    while (answer < 1 || answer > 2)
    {
        printf(Enter 1 for Yes, 2 for No);
        scanf( "%f", answer );
    }
    return answer;
}
```

9. Using Listing 7.1, complete the print_report() function so that it prints the rest of Table 7.1.

10. Write a program that inputs two floating-point values from the keyboard and then displays their product.

11. Write a program that inputs 10 integer values from the keyboard and then displays their sum.

12. Write a program that inputs integers from the keyboard, storing them in an array. Input should stop when a zero is entered or when the end of the array is reached. Then, find and display the array's largest and smallest values. (Note: This is a tough problem, because arrays haven't been completely covered in this book yet. If you have difficulty, try solving this problem again after reading Day 8, "Using Numeric Arrays.")